Navigating a MiDEF file

Scott Millett, Naval Air Systems Command
Network Centric Warfare Directorate, Code 4.0X

This paper is intended as a tutorial for those planning to write editor, interpreter, and parser routines for files complying to MIL-STD-3014, "Department of Defense Interface Standard for Mission Data Exchange Format," known colloquially as MiDEF.

Introduction

Figure 1 shows a MiDEF file's top-level structure. The file has a header followed by data. The header contains the information needed to transport the file to its destination over data links and aircraft data buses, followed by the "Table of contents" information (called the Element List) that is used by file parser routines to determine the contents of the file. Following these two types of information in the header are sets of optional fields, which can be inserted by those who create MiDEF files to support optional file management functions.

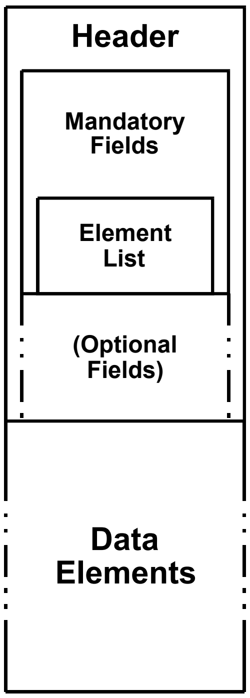

Figure 1 – Top-Level File Layout

Following the header are the data elements that are the conveyed contents of the MiDEF file. There are three classes of data element in a MiDEF file. All are identified by a CLASS code, which identifies them in the Element List. (See Figures 2 & 3.) The CLASS code is a unique identifier for each data type. All CLASS codes are maintained by an on-line government registry maintained by the DoD organization responsible for the MiDEF standard. The registry defines the data type, and is the sole source for allowable values of CLASS codes and their associated definitions, including bit-level definition and size. There are three types of MiDEF data elements:

- Primitives are single data entities of fixed bit size and format. Different primitives can have different sizes, but a given CLASS code will always be associated with one specific type of data entity, with one specific bit size. A latitude might be an example of a primitive. All primitive sizes are in even binary multiples of 8 bits (one byte.)

- Concatenated data types are ordered sequences of primitives. They contain a specified sequence of a specified number of specified primitives. An example of a concatenated might be a three-dimensional coordinate, defined as a latitude primitive, followed by a longitude primitive, then an elevation primitive.

- Module data types consist of a header followed by any combination of data elements, including primitives, concatenateds, and other modules. A MiDEF file is a module. Any module may contain other modules, and may be contained within another module. All rules for working within modules, described herein, apply equally to all modules, whether or not they are mission files.

CLASS Codes

Most CLASS codes used in MiDEF files will be standard types, available for use on all systems. The data entities defined for these CLASS codes can be found in the on-line MiDEF registries located at (http://MIL-STD-3014.navy.mil). MiDEF also supports program-unique CLASS codes for all data types. Programs can request new standard or program-unique CLASS codes by filling out and submitting on-line application forms linked to the appropriate web pages. The open-source information associated with program-unique data entities is minimized. Program-unique CLASS codes support non-disclosed data definitions that may be required for proprietary or classification reasons, while retaining configuration management to ensure that no CLASS code can ever have two different definitions. Each program may request a small number of program-unique CLASS codes; standard data types are expected to satisfactorily supply most of the data entities required for any mission data file, including modules with standard class codes that carry program-unique data elements.

The first bit of all CLASS codes identifies them as modules or not. The rest of the bits of program-unique CLASS codes are assigned by the registering authority to each standard and program-unique data element. Figure 2 shows the configuration of CLASS codes for all data elements.

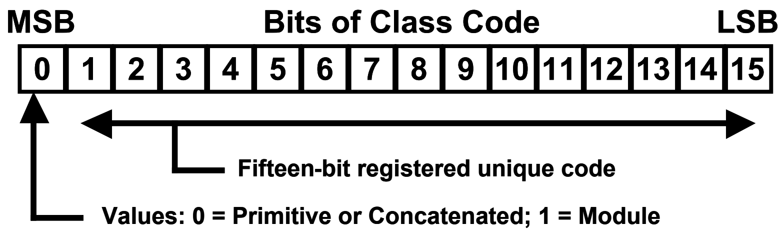
MSB
Bits of Class Code
LSB
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
Fifteen-bit registered unique code
Values: 0 = Primitive or Concatenated; 1 = Module

Figure 2 - CLASS Code Bit Format

Mandatory Fields

Module headers consist of a set of mandatory fields, shown in Figure 4, followed by a set of optional fields, shown in Figure 5. Headers (mandatory and optional fields) are followed by the data elements the Module is conveying, as shown in Figures 1 and 5.

Figure 4 shows the mandatory fields at the beginning of each module, and the placement of the HEADER CONTENTS field within the mandatory fields.

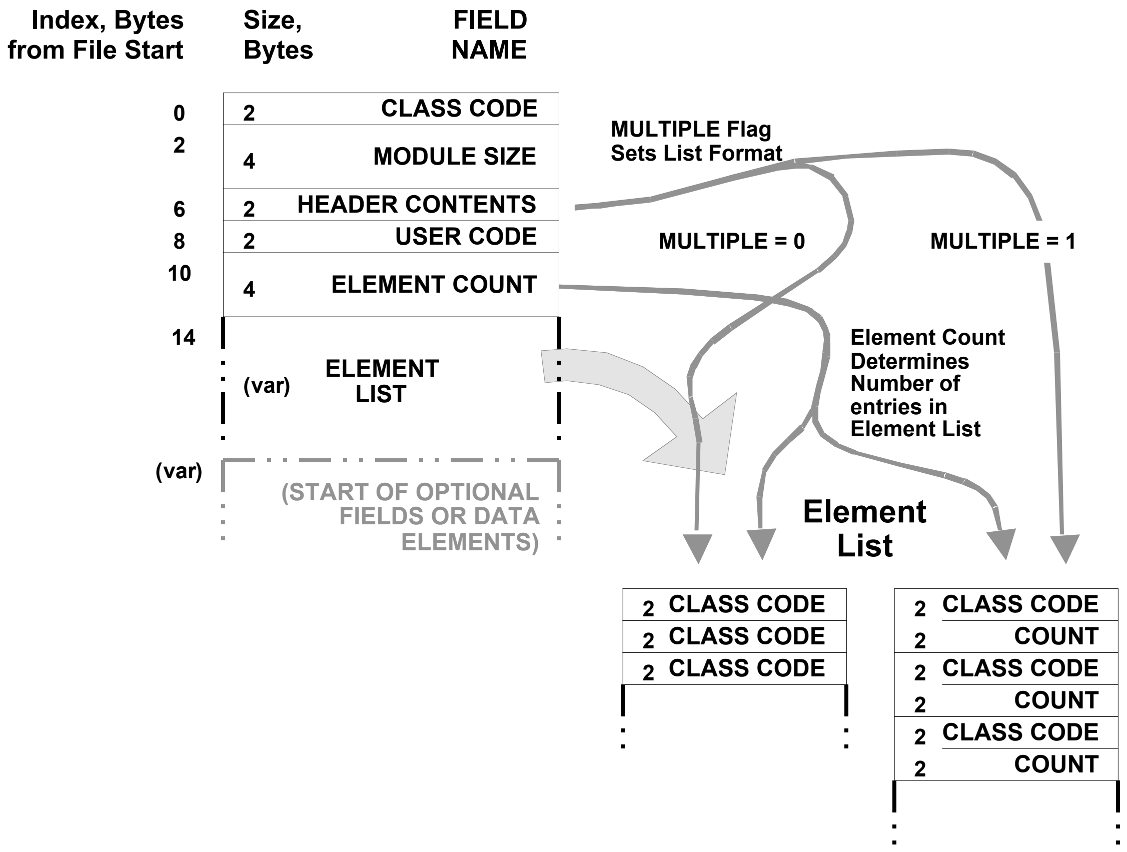

Figure 4 – Mandatory Header Fields, Including Element List

Optional Fields

Figure 5 shows the optional fields, and their relation to the flags in the HEADER CONTENTS field. Optional fields immediately follow the last mandatory field (the Element List) at the end of the module header. The optional fields are followed immediately by the data element content of the module.

Optional fields are groups of special-purpose fields that may or may not be included in a given module. The presence of each set of these fields in any given module is flagged in the mandatory header field called HEADER CONTENTS. HEADER CONTENTS is a field consisting of sixteen one-bit flags. Each flag is linked to a specific set of optional data fields. Unused flags (bits in the HEADER CONTENTS field) are reserved for future use, and may not be used for uncontrolled purposes.

Optional header fields always apply to the total contents of a module, treated as a unit.

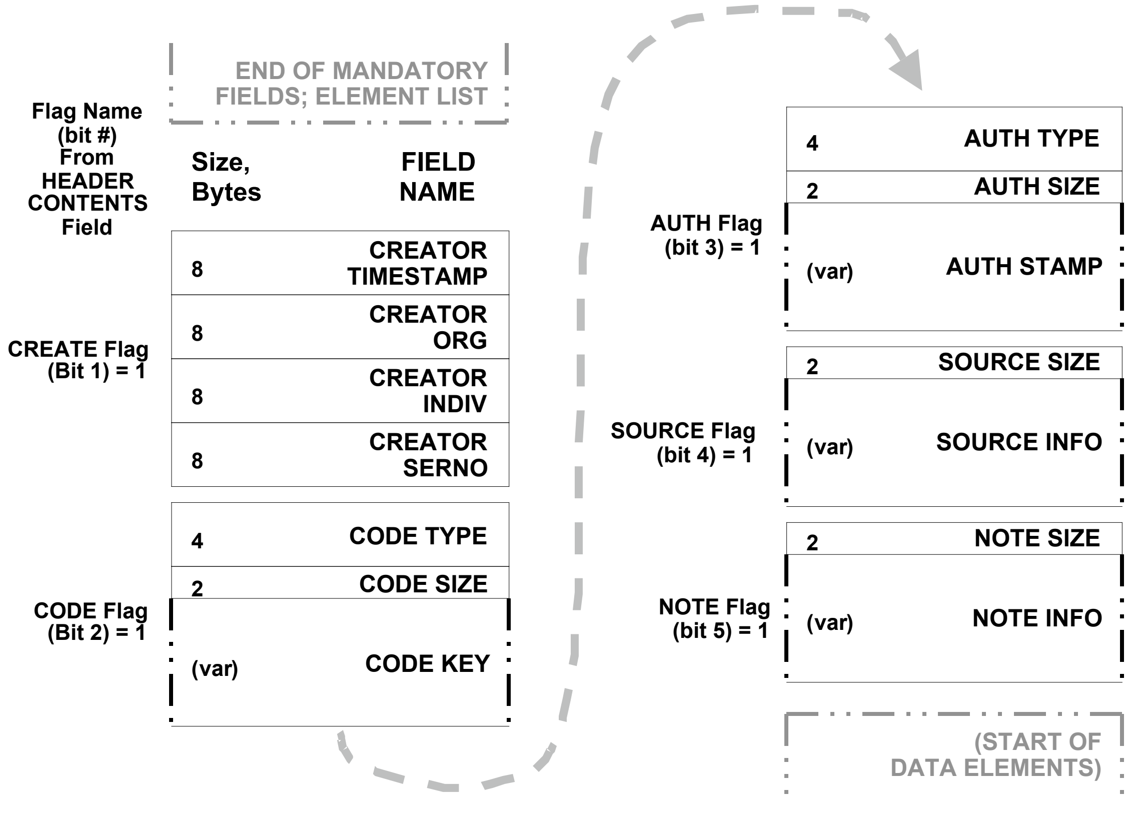

Figure 5 – Optional Header Fields

Optional fields are intentionally general in format, since their use in future, even for their specific purpose cannot be specifically predicted. The website hosting of optional field registries, which support the use (and defined, permissible values) of optional fields, allows the rapid addition of new entries. The use of SIZE fields supports efficient transmission of small entries, simultaneously with the flexibility to carry large entries. The use of a single 16-bit flag field to indicate the presence of all optional fields ensures that users who don't need them pay a small price for the capability they don't use. The optional field groups defined below use only five of fifteen available bit flags, so there is large potential for unanticipated future uses of the optional field capability.

Briefly described, the Optional Field Groups are described below:

> CREATOR Fields identify the organization, individual, serial number and date/time of creation of the module. These fields will typically be used for files generated in contingency planning, where the traceability chain of a module is important. When the module is to be used immediately, sent from a creator directly to the user, the value of these fields may not justify the cost of their transmission and interpretation.
>
> CODE Fields support the mass-encoding of the data element set following the header in this module. The fields allow multiple encoding or encryption schemas, and the forwarding of public keys, checksums, or other ancillary data to support decoding or decryption. These fields are independent of any such capability that is available in any individual communications channel or protocol. Nested encoding/encryption of an entire file or selected modules is supported. This capability may be used to support multi-level security strategies.
>
> AUTH fields support authentication, validation, and digital signature strategies, ensuring trusted sources and supporting network defense against spoofing, spamming, and other attacks. These fields are independent of any such capability that is available in any individual communications channel or protocol. Nested authentication of an entire file or selected modules is supported.
>
> SOURCE fields support traceability of derived files/modules to the source files/modules or other data from which they were generated. By allowing the copying of the CREATOR fields (or other traceability information) from the source data, especially in contingency planning, the use of SOURCE fields can support the quick propagation and impact assessment of updated information to all affected mission files.
>
> NOTE fields allow the transmission of descriptive information to be read by human operators, allowing for simplified human-machine interfaces on MiDEF receiving units. Such units may implement use of the NOTES fields as an alternative to fully interpreting such files, as a means to allow operator/pilot understanding of file contents.

Purpose of CLASS Codes
CLASS Codes are 16-bit codes used to identify specific data element types, so they can be correctly interpreted by receiving software. Only the first bit of each CLASS code has a specific meaning; a value of "1" indicates that the data element is a module; a value of "0" indicates that the data element is not a module (it is either primitive or concatenated.)

The CLASS Code is intended to support the receiving software by providing an index for a suitable receiving handler routine for that data element.

CLASS Codes and their associated definitions can be found on the MiDEF Registry website (http://MIL-STD-3014.navy.mil).

The Element List

Located at the end of its header's mandatory fields, a module's Element List consists of the CLASS codes of each data element in the module, in the same order as those data elements. (See Figures 1 & 4.) The structure of the Element List is defined by two other fields that precede it in the Mandatory fields section of the header. Bit 0 of the HEADER CONTENTS field is the MULTIPLE flag, which determines whether the Element List is a simple sequence of CLASS Codes (MULTIPLE = 0) or a sequence of field pairs, each CLASS code being followed by a 16-bit integer COUNT field indicating the number of instances of that class code to be found in sequentially in the data elements (MULTIPLE = 1.) This option supports compact, efficient element lists, whether the contents of the module are sets of different, mixed data types (Multiple = 0) or large repeating numbers of the same data type (Multiple = 1.) The latter case is typified by routes with many waypoint concatenated elements, imagery with many pixels primitives, or templates with many line segment concatenated elements.

The ELEMENT COUNT field is the other mandatory header field that influences the format of the Element List. Its value defines how many CLASS codes (or CLASS code and COUNT pairs when MULTIPLE = 1) will be found in the ELEMENT LIST.

When is a Module a File?

The term "File" is applied to a module when it is separately stored, used, or transported from one computing environment to another. "File" is a functional definition unrelated to any specific internal feature of the module. To paraphrase the old entomologist's mantra regarding bugs and insects: "All MiDEF files are modules, but all modules are not files." Modules are files when they leave a MiDEF editing environment without being embedded within a higher-level module, and when they enter a software environment that forwards them or uses their data contents.

For example, a complete mission data set, defined in a module, would be a single file if it was sent by the editor software to another piece of software, or if it was saved to disk.

Even a complete mission data module would not necessarily be a file, if it was saved or transported only as a part of a set of mission data modules.

On the other hand, a route module is not a complete mission data set, but such a module could be saved to disk (as a route file) for shared use by several weapons to be launched on one mission to different aimpoints in one target area.

Common Mission Data File CLASS Codes

CLASS codes for MiDEF modules at the mission data file transfer level will generally fall into one of these three categories:

- MSN PLAN – The mission plan module defines a complete mission plan for a system. It contains all necessary data from the mission planning function to enable a computer-based system to carry out a desired task.
- MSN UPDATE – The mission update module defines a set of data intended to update an existing mission plan. This will typically be a partial set of mission data, such as a new target coordinate and velocity vector. (The "SOURCE" set of optional header fields is recommended for use by the MSN UPDATE as a means to allow the system accepting the data to determine which of multiple plans is to be modified by the MSN UPDATE file.)
- MSN SET – The mission set module is a set of mission files. Its data elements are MSN PLANs and/or MSN UPDATEs. The mission set allows grouping of multiple missions into one convenient package for transfer to the system that will use them. Possible applications will be:
    - Multiple alternate preplanned missions intended for a single weapon
    - A new set of missions for all of a platform's weapons, in support of mission flex because the platform had to take over the mission of another that was unable to complete its mission
    - A coordinated set of new weapon mission(s) plus aircraft mission updates (waypoints, tanking, etc.) to support delivery of the new weapon mission. (This example could be used by a weapon platform that was able to accept its own mission updates via MiDEF.)

Other Module CLASS Codes

Module CLASS Codes that are typically used within a file are intended purely to aid the editor in telling the user system how to correctly interpret the structure of a received data file. The file size overhead cost of subordinate module headers may be a good investment if they can greatly simplify the interpretation and/or forwarding of a file. For example, a weapon built by a true system integrator prime might receive a mission file grouped into:

- A route plan module, which defines the number, sequence, location and characteristics of waypoints, forwarded to an INS subsystem.
- A terminal profile module, which defines the terminal flight path heading, altitude, and range to target for seeker acquisition, plus desired impact velocity, heading and dive angle, sent to a terminal autopilot routine and also forwarded to a seeker subsystem.
- A fuzing module, which groups all the data to be sent to the fuze(s), such as proximity sensing altitude, arming range-to-target, impact delay, cavity counts, and so forth, forwarded to a "smart fuze."
- A terminal reference module, which contains the reference used by the seeker to recognize the desired aimpoint, forwarded to a seeker subsystem.

Definition of such sub-modules would greatly facilitate the development of a system of systems, by isolating the mission data format and content associated with separate internal ICDs that impact different subcontracts.

Other module types initially available include:

NAV	ROUTE	Route plans
WHD	DATA	Fuzing data
APT	DATA	Aimpoint information, including terminal profiles
NAV	DATA	General nav data, which can include route and aimpoint
LNK	DATA	Data link setup and operation data
TGT	DATA	Target data, including seeker reference
MSN	WRAP	A wrapper for a legacy-formatted weapon data file
MSN	PLTFM	A platform-level grouping, which can include multiple weapon types and/or platform mission data

No subordinate module is required in any data file. They are available strictly to aid developers designing weapon mission planning and operational flight program software.

Types of MiDEF Receiving Software Routines

This section addresses the software routines on computers that receive complete MiDEF files, such as aircraft and weapon mission computers. It is assumed in this paragraph that the communications channel transfer operation has been completed, and that an error-corrected, reassembled file, stripped of channel-specific header/trailer content, has been delivered to the computer, with a cue to trigger the receiver software discussed herein.

Based on their purpose and the extent to which they have to delve into the MiDEF file structure, receiving software routines can be broken into three categories:

- <u>MiDEF Transfer Routines</u> have the simplest task; to transfer a MiDEF file to another destination. Host computers that are connected to data link terminals that transmit MiDEF files are in this category, as are mission computers on platforms that implement the simplest possible MiDEF interface.

MiDEF Transfer Routines need only read two fields to transport any MiDEF file. Each field is a fixed index (offset in bytes) from the front of the file. The USER field identifies the intended destination of the file, and the MODULE SIZE field tells how big it is. Although these fields are not directly required by some comms channels and system architectures to enable communications, because other means of identifying destination and file size are available, these MiDEF header fields are available for use by the host computers of transmitting, receiving, and transporting elements of any communication chain. Caution should be exercised in making the decision that they need not be filled in for any particular application. MiDEF is specifically intended to support general-purpose, long-lived architectural design, and its implementations should be made as general-purpose as possible from the start, to minimize cost of integrating new systems.

- MiDEF Parser Routines read the element list (the “table of contents”) of a file that contains module elements, then break out each module based on its intended destination, and send that module to either a transfer routine for transport to another computer, or to an interpreter routine, if the module is intended for use by that computer and its host. Parser routines are typically used on platforms that receive MiDEF files of the “mission data set” type, intended for multiple weapons, or for systems that have multiple computers performing different functions, such as a platform using MiDEF format that might send a MiDEF sub-file to a navigation computer, another to its Stores Management Computer, and yet another to a sensor subsystem.
- MiDEF Interpreter Routines read MiDEF files/modules in their entirety. Interpreter routines are the ultimate destination of every MiDEF module; the actual consumers of mission data. These routines process the full extent of MiDEF files. Interpreters use the element list in a module to sequentially extract the data from the module and, keyed by the type of element as defined by its CLASS code, perform the necessary activities such as data storage in specific locations, calling of subroutines, or initiation of software events. Interpreters will typically have the ability to read and manage nested sub-modules; and for that reason, the parser capability is often included in an interpreter. Weapons will typically need only an interpreter, but platforms that use MiDEF for their native data might integrate the interpreter and parser functions, and will include a file transfer function in the interpreter, to allow portions of mission data set MiDEF files to be forwarded to stores and other subsystems.

Identifying a Module’s Class and Contents

A MiDEF module starts with its 16-bit CLASS code. This code can be used to confirm or determine the purpose of the module. The receiving routine (interpreter, parser, or transfer) may be branched at this point based on the CLASS code. The CLASS code is selected from recognized values in the CLASS code registry.

MIL-STD-3014 requires that compliant receiving software have a way to process modules and data elements whose CLASS codes are not recognized. It is recommended that systems designed to use MiDEF should implement a feedback process, whereby the recipient alerts the sender when it is unable to successfully interpret and accept any portion of a received MiDEF file.

The class code is followed immediately by a 32-bit integer, the MODULE SIZE field, which defines the modules size in bytes. Maintaining a correct value in this field for the module being edited, and for all levels of superior module containing it, is the responsibility of any function that edits a module.

The MODULE SIZE field is followed by the 16-bit HEADER CONTENTS field, which indicates the presence of optional elements of the header. These optional elements are ordered within the module header in the same sequence as their flags in the HEADER CONTENT field. Bit position 0 affects the format of the element list, and is discussed in the following section. Bit positions 1 through 15 serve to identify the presence of sets of optional fields within the header. Optional fields contain notes, source references, digital signatures, encoding or encryption data, and the like, but the fields themselves are not present in any given MiDEF module unless they are required, in which case their presence is indicated by the assertion of the appropriate bit in the HEADER CONTENTS field. Some of these sets of optional fields have one or more fields of variable size. In all cases where a variable-size field is present, that field is immediately preceded with a 16-bit integer "SIZE" field, defining (in bytes) the size of the variable-size field.

- Note: To keep minimal implementations of MiDEF receiver software as simple as possible, all optional field sets (not counting the MULTIPLE flag) may remain unimplemented by system convention. A simple bit mask test to ensure that bits one through fifteen are disasserted (set to zero) by the receiver can confirm this status in the received module. With that confirmation, the header is known to end with the element list. Prior to the element list, and including the ELEMENT COUNT field (see below,) a header with no optional fields is14 bytes in size, and the data elements will immediately follow the element list.

In order to properly account for the possible presence of optional fields, any interpreter, in addition to processing the fields themselves, must establish an indexing strategy that allows the set of fields to be accounted for when locating the beginning of the module's data elements. As can be seen in Figure 5, the value of the index for each field set is the sum of the sizes of the fixed fields, plus the value found in any embedded XXX SIZE field(s,) each of which defines the size of the variable field immediately following.

The ELEMENT COUNT Field and the Element List

Following the optional fields in the header is the Element List, or "table of contents" for the data carried by the module. This table of contents consists of the 32-bit integer ELEMENT COUNT field, followed by the Element List. The number of entries in the

Element List (which correlates the number of data elements in the module) is indicated by the value in the ELEMENT COUNT field.

The Element List can have two formats, as shown in Figure 4. The first format is a simple list of 16-bit CLASS codes, one for each data element contained in the module. The second format for the Element List is a sequence of field pairs, each pair consisting of a CLASS code followed by a 16-bit integer COUNT field that defines the number of sequentially repeated elements that are defined by that CLASS code. This second format is useful when the module contains large numbers of similar data elements, such as waypoints, image pixels, or legacy file blocks.

The 0^{th} bit of the HEADER CONTENTS field, known as the MULTIPLE flag, defines which of the two formats is used in the Element List. A MULTIPLE value of 0 (negated) indicates the first format, in which each class code in the Element List represents a single data element following the header. A MULTIPLE value of 1 (asserted) indicates the second format, in which each class code in the header list can represent multiple data elements in the module.

Legal values of COUNT fields can be zero or any positive number. Values of zero in COUNT fields represent unnecessary overhead; they represent an absence of the associated data element, and thus are just wasted space. Nonetheless, values of zero are permitted, and designers of parsers and interpreters need to design them to properly handle COUNT values of zero.

The MULTIPLE flag is asserted for the entire module. When MULTIPLE is asserted, every data element needs a COUNT field following every CLASS Code. COUNT field values of 1 will be common, since many modules that carry large sequences of repetitive data will also carry several individual support data elements, such as fields defining the number of pixels in a line and the number of lines in the image, in a pixel-based image file.

The total size of the header is:
- The size of the mandatory fields before the Element List (14 bytes,) plus
- The value of the ELEMENT COUNT field multiplied by 2 bytes (MULTIPLE flag negated) or 4 bytes (MULTIPLE flag asserted), plus
- The total of sizes of any optional XXX SIZE fields that are present

This value can be calculated from the header data alone, by the processes described above.

Interpreting Data Elements Carried Within a Module

Immediately following the last entry in the Element List of the header is the first data element carried by the module. The type of element is indicated by the class code value of the first entry in the Element List.

An obvious interpreter approach is to define lookup tables whose entry columns are CLASS codes. Two such tables are recommended, one for primitive and concatenated data elements, and another for modules. This is because the size and specific content of primitive and concatenated data elements is determined solely by their class code, whereas module data elements have variable length that must be calculated by recursion of the process being described in this document (such as a stack-based process.) Also, subordinate modules are likely to be forwarded to other software modules by interpreters.

A bit test on the 0th bit of any CLASS code in an Element List can quickly determine whether that element is a module.

For primitive and concatenated data elements, each entry in the lookup table could be a CLASS code followed by the size of the element in bytes, and then a pointer to the routine that processes the element, or the address at which the element is stored.

It should be noted that processing routines for these simpler elements may not themselves be simple. For example, consider the routine that processes a concatenated data element such as a lat/long/elev coordinate. The coordinate defined by that CLASS code may define a launch point, a waypoint, a target, or any of a broad variety of other possibilities, depending on the context of the element's location in a mission data file. Typically, there will be an agreed implementation between a tactical system that acts upon the MiDEF file, and the mission planning system that creates it. A context-based approach to CLASS code interpretation, which modifies the lookup table depending on the module in which the element is found, may be a useful architectural construct. The processing routine for the module that carries the coordinate data would then modify the lookup table or the handling routine with other, context-based information, telling the coordinate processing routine to treat this particular coordinate is a target or waypoint, etc.

Since MiDEF intentionally imposes no limit on the levels of embedded modules, it is possible to build MiDEF files that are many levels deep, by creating new subordinate modules to carry data in a very hierarchical fashion, or by grouping existing modules into new higher-level modules. This strategy is probably not useful in a final tactical mission data file, due to file size and receiving software complexity, but may be of valuable in parallel or contingency planning, prior to a final scrub to ready the file for transport to the user system. In any case, interpreter and parser software is likely to have much greater long-term durability if it supports recursive handling of multi-level nested modules.

The Benefits of Concatenated Data Elements

Since each instance of a module carries overhead both in file size and in processing complexity, a reasonable balance must be achieved between modules and primitive data elements. The concatenated data element is a powerful tool in this regard. Where primitive data elements are regularly grouped together in fixed relationships, a concatenated element can be defined and used in lieu of a module.

A good way to think of the concatenated data element it that it's is a fixed-format module that doesn't need a header. Therefore, it carries no more overhead than a primitive, and can carry any amount of data, so long as the count and order of primitives is fixed. Coordinates are excellent examples of concatenated data elements. There are several available formats to choose from, 2D (Lat & Long,) and 3D (Lat, Long, & el,) with and without accuracy estimates (Lat, Long, El, CE, LE,) etc. Additional standard or program-unique CLASS codes can be quickly and easily issued to meet future needs. A digital video raster line and a seeker template line segment are other possible examples.

If a user has a programmable fuze, for example, and it always takes the same number of inputs in the same order, then the entire string of fuze data can be defined as a single concatenated data element.

Similarly, no matter what information you might put into waypoint data for your system, if it's the same data set for each waypoint, you can make that data set into a concatenated data element. You might take a 3-D coordinate, add a velocity vector through that point, a new antenna pointing direction on the next leg, weather conditions expected on the next leg, and so forth. Define the sequence of all that data as a concatenated element, call it "Waypoint" internally, request the assignment of a program-specific CLASS code through the website, and your route plan can simply be an embedded series of "Waypoint", "Waypoint", "Waypoint", "Waypoint",…, as many as you need for a particular mission. You don't need a separate module or even a contiguous sequence in your mission plan module, just an agreed convention between planner and weapon that the order of waypoints in the file corresponds to the order of waypoints in flight.

The limitation of a concatenated element, when compared to a module, is flexibility. It can't carry variable amounts of data, nested elements, notes, or a destination address, etc. It is simply a useful compromise.

Use of Generic Mass Data Transfer for MiDEF files

MiDEF enables a departure from the legacy practice of "customized" mad data transfer, in which the transfer protocols of the communications channel (message number and message size on the 1553/1760) bus is tied to, and influenced by, the amount and content of data to be transferred. This legacy practice means that, since the bus message number is used to communicate to each weapon the size and content of the message data conveyed, each change in size or content of data transmitted to legacy weapons has an impact on the bus communication software. This is one of the reasons that updates to weapon software that require modification to mission data files requires changes to platform OFP software.

By evolving to a generic file transfer practice, in which ALL mission data (entire files and updates) for new, MiDEF-compliant systems is transmitted and received as a variable length MiDEF file, a only a single, unchanging communication protocol ("Here's 'n' blocks of MiDEF mission data for you") is required, regardless of who's sending, who (MiDEF-compliant) is receiving, what content is in the message, or how big it is.

Because of the flexibility and capability of the MiDEF file/module header, the receiving system can figure out what it has received from the file itself, and does not need any "coaching" from the comms channel, such as unique message numbers or fixed message sizes.

It is strongly urged that anybody implementing the communication of MiDEF files over any hard-wired or radiated communications channel, implement that transfer as a generic mass data transfer of the MiDEF file, that will allow any sender to send a MiDEF file of any size to any receiver, with a content that is uninfluenced by, and has no influence on, the comms channel or the processes and protocols of communication.

A Legacy "File Wrapper" Interpreter Approach for Weapons

This section deals with a strategy for implementing a minimal platform software change that allows a legacy weapon to receive mission data in its legacy file format, while allowing the platform and external communication media such as tactical data links, to implement an inclusive, generic MiDEF file transfer strategy.

A brief review of the 1760 mass data transfer process, as it applies to a MDT update to a previously stored mission, is in order here. 1760 uses the Transfer Control (TC) message, numbered 14R, to set up and control the progress of mass data transfers. the Transfer Data (TD) message actually carries the 29- or 30-word blocks of mass data in the MDT process.

MIL-STD-1760 allows TD messages for receive (download) or transmit (upload) to use message numbers (subaddress select field values) chosen by the user from among the following user-defined values (See Table B-1 in MIL-STD-1760, Appendix B): 2-6, 9-10, 12-13, 15-18, 20-26, and 28-29. All of the current set of 1760 weapons use the 13R message number for some form of target data transfer, and so in the following discussion, the use of "13R" is generic, and the concept can apply to any legal TD message number.

The 14R TC message performs several functions:

- Erase a record, or a file, or all files. As implemented currently on weapons, a file often represents a category of mission data, and a record represents one of several selectable numbered missions within that category. The complete set of data required to complete a particular mission can span several files.
- Enter (or remain in) either the download mode (data to weapon) or upload mode (data from weapon.)
- Start processing a new file of a specific file number, or a new record number within the current file.
- Initiate a message-based or block-based (embedded) checksum mode
- Exit mass data transfer (upload or download) mode

A 13R TD message always transmits 30 16-bit words, wrapped with associated control and header words, and coordinated with a status word that leads the data words (when the weapon confirms its readiness to transmit the data) or follows the data word (when the weapon confirms its receipt of the data.) The first data word is always a concatenation of the record and block numbers of the 28 or 29 file content words that follow. If the MDT implements a block-based embedded checksum mode, the last word is a checksum; otherwise it is the 30th data word (a.k.a., the 29th content word.) In the last block of any MDT, even if all of the data words in a block are not needed, they are sent, and discarded by the receiving software.

This strategy exploits characteristics of the legacy interface, and uses 14R Transfer Control, plus 13R Transfer Data messages and the flexibility of the MiDEF format. The basis for this strategy is that:

- All legacy mission data can be sent to the weapon as one of the several files, records, and blocks in a 13R (or equivalent) mass data transfer message.
- The 13R message identifies each block with its record and block number (in accordance with the 1760 standard,) and the record/block number identifies the data being sent.
- The ICD for each weapon defines the subaddresses, options and protocols it implements in the Transfer Control and Transfer Data messages.

MiDEF offers three data types that support efficient transmission of legacy data files

- A concatenated data type called “MSN LGCY TD BLOCK” that that starts with a 16-bit primitive “BUS SUBADDR,” and concludes with 30 words (60 bytes) of data that represent words 1-30 of a legacy block as transferred under the 13R/T message. The receiving platform reconstructs the actual stores bus 1760 message by wrapping this data with fixed boilerplate values.
- A concatenated data type called “MSN LGCY TD MSG” that contains six words (twelve bytes) of data that represent words 2-7 of a 14R Transfer Control message. The receiving platform reconstructs the actual stores bus 1760 message by wrapping this data with standard command, header, and checksum words according to the standard stores bus rules.
- A concatenated data type called “MSN NULL TD BLOCK” that that starts with a 16-bit primitive “BUS SUBADDR,” and concludes with a GENERAL PACKED BYTES data word that defines the values of record and block number that is located in the first data word of every 1760 MDT TD message. The receiving platform reconstructs an actual stores bus 1760 block full of null data by wrapping this data with fixed boilerplate values, including 28 words (56 bytes) of null data, followed by either a 29th word of null data or, if embedded block checksum mode is enabled, a checksum word.

These three data types can be created in sequence through a scan of a legacy file, to generate, in accordance with each weapon's ICD, an extracted MiDEF data file.

To protect against inadvertent mixing of new updated data in some file blocks with obsolete data in others, appropriate operationally vetted file editor software safeguards and user training must be emplaced. This can be achieved during the file translation software creation process, with any chosen combination of TC file and record erasure commands, and TD null block data elements, to either pre-erase or over-write blocks related to the mission being updated, to ensure that residual data from an obsolete mission does not remain embedded within the mission program space on an updated weapon.

A notional reprogramming sequence, based on replacement of the third mission on a legacy "J-weapon," might occur as follows:

1) Platform receives a MiDEF-compliant mission data file update over a communications channel.

2) Platform activates pilot interface, displaying update description, including weapon type for which the mission is valid, and store stations containing that weapon type.

3) Platform accepts pilot acceptance/rejection of the update, and pilot designation of the specific store and mission number to be replaced/updated by the newly-received mission data.

4) Platform performs pre-erasure of obsolete data for the mission to be updated. Note that for many legacy weapons, mission data is grouped in several files. Some files are common to all missions (such as GPS crypto keys), and other files contain categories of data that is unique for each individual mission. Therefore, the data to be replaced to update mission "n" is located in several different files, as record "n" in each of those files. The pilot's selection of mission "n" generates a programmed sequence of 14R Transfer Control messages that instruct the weapon to erase record "n" in the appropriate files on the weapon.

5) Platform confirms erasure and readiness to accept new data, and then places the weapon into the mode to accept new data via mass data transfer.

5) Platform reads a sequence of LGCY TD BLOCK concatenated data elements from the MiDEF file, and uses the BUS SUBADDR sub-element to build the command word.

6) Optional: For each LGCY TD BLOCK, if the ICD has determined that the transfer incorporates the block-embedded checksum protocol, the platform reads in all of the data words from the LGCY TD BLOCK concatenated data element, calculates the checksum based on the constructed command word and the all data words but the last, and replaces the last data word with the calculated checksum. This calculation must be performed

following the pilot designation of the store station, because that selection impacts the TD command word and therefore the checksum.

6) For that data block, the platform sends the constructed command word, followed by the data words (including the optional embedded checksum if applicable) to the weapon.

The complete sequence might notionally look like this, from the platform's perspective:

- Instruct store to switch to download mode.
- Loop
    - Pause for a timeout period.
    - Request TM message. If Transfer Mode is not 8000H, repeat steps 2 & 4 one time.
- Instruct store to erase memory.
    - Loop
        - Pause for a timeout period.
        - Request TM message until = 8800H.
        - Max erase time is a specified timeout period.
- Instruct store to start new file/record.
    - Loop; one repeat maxium
        - Pause for a timeout period.
        - Request TM message. If Transfer Mode is not A000H, repeat.
- Transfer Data
    - Loop for each file/record to be transmitted
        - Send transfer data messages until all blocks are sent
        - Instruct store to perform checksum.
            - Loop
                - Pause for a timeout period
                - Request TM message until Transfer Mode = 8100H or 8180H.
                - Max checksum time is a specified timeout period.
- Instruct store to exit transfer mode.
    - Loop
        - Pause for a timeout period.
        - Request TM message. If Transfer Mode is not 0000H, repeat.

The decision to generate the appropriate TC messages entirely within the platform, or to pre-calculate them on the ground and include them in the MiDEF file, and how to handle data integrity with respect to residual data, can be made by the implementers of the MiDEF file editor (creator) and interpreter (user.)

In making this decision, implementers should consider the impacts of multiple, inconsistent implementations for different platforms, weapons, and mission data file generation systems. While MiDEF can support a large number of implementation strategies, a single, Joint, cross-platform, cross-weapon, and cross-workstation strategy, consistent across the entire DoD and international weapon integration community.

This is only a recommendation; the decision is up to the operational user communities who set requirements and provide user feedback to developers, and the development community who must deliver effective products in a resource-constrained environment. MiDEF will support both interoperable and stovepiped solutions, but strongly favors the coordinated, interoperable solution.